Managing Data-Movement for Effective Shared-Memory Parallelization of Out-of-Core Sparse Solvers

Haim Avron
IBM T.J. Watson Research Center
haimav@us.ibm.com

Anshul Gupta
IBM T.J. Watson Research Center
anshul@us.ibm.com

***Abstract*—Direct methods for solving sparse linear systems are robust and typically exhibit good performance, but often require large amounts of memory due to fill-in. Many industrial applications use out-of-core techniques to mitigate this problem. However, parallelizing sparse out-of-core solvers poses some unique challenges because accessing secondary storage introduces serialization and I/O overhead. We analyze the data-movement costs and memory versus parallelism trade-offs in a shared-memory parallel out-of-core linear solver for sparse symmetric systems. We propose an algorithm that uses a novel memory management scheme and adaptive task parallelism to reduce the data-movement costs. We present experiments to show that our solver is faster than existing out-of-core sparse solvers on a single core, and is more scalable than the only other known shared-memory parallel out-of-core solver. This work is also directly applicable at the node level in a distributed-memory parallel scenario.**

I. Introduction

Direct sparse linear solvers are robust and typically exhibit good performance, but often require large amounts of memory due to fill-in of the factors. For direct solver algorithms, the use of parallel processing tends to increase the memory requirements, so the recent shift towards manycore CPUs further aggravates the problem. Even though the amount of memory available in modern machines has been increasing, it cannot keep pace with the growing memory requirements of direct solvers due to larger and larger linear system being solved in many applications.

One possible solution is to use a preconditioned iterative method. In many cases, a properly chosen (problem specific) and tuned preconditioner will be highly effective for certain linear systems. Yet development, programming, and performance tuning of such preconditioners on modern HPC platforms is time consuming and challenging. Direct methods are usually much faster than using a poorly suited preconditioner. The robustness and good performance of direct methods are the reasons why direct methods are often used in many industrial applications, such as structural analysis, electronic design automation, device simulation, and computational design optimization for aerospace, energy, and manufacturing, etc.

The use of out-of-core (OOC) techniques, that is keeping some of the data on secondary storage, is a popular method for mitigating the memory requirements of direct methods for solving very large systems. Unfortunately, using OOC techniques introduces a fair amount of data-movement overhead and serialization to the process. The goal of this paper is to analyze these additional costs, and propose ways to overcome them. More concretely, the contributions of this paper are as follows:

- We analyze the data-movement costs and various trade-offs in a shared-memory parallel OOC solver. This analysis, which reveals some surprising conclusions, motivates our shared-memory parallel OOC algorithm.
- The algorithm based on our analysis uses a novel memory-management scheme and adaptive task parallelism to reduce data-movement costs. Our memory-management scheme allows a large portion of the available main memory to be utilized for storing the factor matrices while keeping I/O overhead low. An adaptive task-allocation scheme determines the best balance of fine- and coarse-grain parallelism based on the size of factors and available memory.
- We implemented these techniques in a new parallel OOC linear solver software for symmetric positive definite matrices. To the best of our knowledge, ours is one of the only two shared-memory parallel sparse OOC solvers, and we present experimental results to show that it is significantly faster, more scalable, and more robust than the other.

In the context of a sparse OOC solver, the issues related to shared-memory parallelism tend to be more complex than those related to distributed memory. On a distributed-memory system, dedicated secondary storage on each node or a parallel file system can provide natural means of developing OOC extensions to an existing parallel in-core solver. However, a parallel OOC solver on a multicore/multiprocessor node of a larger distributed-memory system or on a stand-alone shared-memory parallel system must overcome the constraints imposed by an effectively serial access to secondary storage (Section IV-A). In this paper, we focus on shared-memory parallelism. Both our analysis and algorithmic techniques are applicable to the task-subgraphs of a parallel multifrontal solver being handled on individual nodes of a distributed-memory machine.

SC12, November 10-16, 2012, Salt Lake City, Utah, USA
978-1-4673-0806-9/12/$31.00 ©2012 IEEE

The remainder of the paper is organized as follows. Related work is described in Section II. In Section III, we give the necessary background on the OOC extension to the multifrontal method. In Section IV, we analyze data-movement costs in an OOC multifrontal solver. Section V describes our algorithm. Section VI contains experimental comparisons of our solver to other OOC solvers. Section VII contains concluding remarks and a brief description of our plans to enhance the impact of this work in the near future.

II. Related work

Over the years, many OOC sparse direct solvers have been proposed [2], [3], [4], [5], [10], [12], [13], [24], [26], [28], [29]. We now review some of the more recent work.

Rothberg and Schreiber [28] investigate several OOC extensions to the multifrontal and left-looking approaches for matrix factorization. They discover that these suffer from excessive I/O for large problems. They introduce several improvements, and report that a left-looking method that uses a multifrontal strategy in some portions of the factorization process is the most effective.

Rotkin and Toledo [29] point out that the multifrontal method suffers from a poor computation-to-I/O ratio. Therefore, they propose an hybrid right/left-looking method with a sophisticated factorization schedule. Their algorithm is very robust, and it can factor very large matrices on machines with a very low memory capacity.

Dobrian [12] compared experimentally and analytically (on model problems) various strategies for factoring sparse matrices: left-looking, right looking and multifrontal. He claims that the multifrontal method works well even when the size of main memory is small. However, he observed that there are cases where it performs poorly, and conclude that a solver should provide several algorithmic options.

Scott and Reid [26] developed an OOC solver based on implicit memory management. Their solver, which uses a multifrontal method, accesses memory in pages. The pages can either be in main memory or on secondary storage. A virtual memory library [25] is responsible for determining which pages are kept in main memory and which ones on secondary storage. The solver itself is oblivious to that decision.

Liu [20] suggested an algorithm to determine the tree traversal that minimizes the working storage requirements of the multifrontal method. He claims that his algorithm should reduce I/O when some data has to be processed out of core. Agullo et al. [5] suggest minimizing I/O directly, claiming that minimizing storage requirements, as suggested by Liu [20], can still lead to excessive I/O volume. They show how to process the task dependency graph of a multifrontal algorithm in a way that minimizes I/O.

Agullo et al. [3], [4] conducted a preliminary study of the issues involved in implementing a parallel OOC solver. Their solver, which is a part of the MUMPS library [6], stores only the factors on secondary storage. They claim that this enables them to solve larger problems on limited memory machines with reasonable performance. They do not provide a fully parallel OOC solver. Indeed, they label their study as preliminary.

In addition to our solver, Intel MKL PARDISO [18] offers the only other shared-memory parallel sparse OOC solver. No paper or technical report on the algorithm used by the OOC module in Intel PARDISO has been published, so it is unclear how this algorithm operates.

III. The Multifrontal Method and Its Out-of-Core Extension

This section describes the basic structure of a multifrontal OOC Cholesky factorization; additional information on the basic multifrontal method can be found in Liu's survey [22]. Our goal here is two-fold: first, to explain where the use of the secondary storage comes into play; second, to establish terms and notations used throughout the paper. In general, we use the notation introduced by Rotkin and Toledo [29].

The goal of the factorization is to compute the factor L of a symmetric positive-definite matrix $A = LL^T$. Throughout the paper, we assume the matrix A is $N \times N$, and that its columns and rows have been preordered to reduce fill (that is, reduce the number of non-zeros in L). The multifrontal method uses a combinatorial structure called *elimination tree* [30] of A. The elimination tree is a rooted forest (tree, unless A has a nontrivial block-diagonal decomposition), with N vertices. The parent $p(j)$ of j is defined to be $p(j) = \min\{i > j : L_{ij} \neq 0\}$. The elimination tree is useful since it represents dependencies in the factorization method [21].

Modern multifrontal Cholesky algorithms use a *supernodal decomposition* of L [14], [23], [27]. The columns of the factor are split into *supernodes* (or *supercolumns*). A supernode is an ordered subset of the columns of the matrix. The supernodes are disjoint and the order of columns inside each supernode is consistent with a postorder traversal of the elimination tree. We denote the supernodes by $\Omega_0, \ldots, \Omega_{S-1}$ where S is the number of supernodes. There are multiple ways to split the columns to form supernodes [7], [14], but these are not the concern of this paper.

We also associate with supernode j an unordered set Ξ_j of row indices in the subdiagonal block. If $i \in \Omega_j$ then column i of L can be non-zero only in indices that belong to $\Omega_j \cup \Xi_j$. If we denote by $X_{R,C}$ the submatrix of X consisting of rows in R and columns C, then the columns Ω_j of L are fully represented using the dense matrix $L_{\Omega_j \cup \Xi_j, \Omega_j}$. We call the matrix $L_{\Omega_j \cup \Xi_j, \Omega_j}$ the *factor block* of supernode j. We define $\omega_j = |\Omega_j|$ and $\xi_j = |\Xi_j|$.

The multifrontal method works as follows. It is a recursive algorithm, where the recursion is guided by the *supernodal elimination tree,* which guarantees that if i is a child of j then $\Xi_i \subseteq \Omega_j \cup \Xi_j$. The multifrontal method proceeds by factoring all the supernodes. Factoring supernode j has two outputs: the factor block of supernode j ($L_{\Omega_j \cup \Xi_j, \Omega_j}$), which is part of the output of factorization, and the *contribution block* (or *update block*) $U^{(j)}$ of supernode j, which is intermediate data used only during the factorization. To factor a supernode j, we first recursively factor all the children of j in the supernodal

Algorithm 1 Template of an out-of-core supernodal multifrontal sparse Cholesky.

1: **ooc-mf-factor**(A, T):
2: **for** each root j in T **do**
3: call **recursive-ooc-mf-factor**(j, A)
4: **end for**
5:
6: **recursive-ooc-mf-factor**(j, A):
7: **for** each child k of j (possibly in parallel) **do**
8: $U^{(k)}_{\Xi_k,\Xi_k} \longleftarrow$ **recursive-ooc-mf-factor**(k, A)
9: mark $U^{(k)}_{\Xi_k,\Xi_k}$ as *evictable*
10: **end for**
11: $\Gamma_j \longleftarrow \Omega_j \cup \Xi_j$
12: $F^{(j)}_{\Gamma_j,\Gamma_j} \longleftarrow 0$
13: $F^{(j)}_{\Gamma_j,\Omega_j} \longleftarrow F^{(j)}_{\Gamma_j,\Omega_j} + A_{\Gamma_j,\Omega_j}$
14: **for** each child k of j **do**
15: retrieve $U^{(k)}_{\Xi_k,\Xi_k}$ if previously *evicted*
16: extend-add $F^{(j)}_{\Gamma_j,\Gamma_j} \longleftarrow F^{(j)}_{\Gamma_j,\Gamma_j} - U^{(k)}_{\Xi_k,\Xi_k}$
17: discard $U^{(k)}_{\Xi_k,\Xi_k}$
18: **end for**
19: factor $F^{(j)}_{\Omega_j,\Omega_j} = L_{\Omega_j,\Omega_j} L^T_{\Omega_j,\Omega_j}$
20: solve $L_{\Xi_j,\Omega_j} L^T_{\Omega_j,\Omega_j} = F^{(j)}_{\Xi_j,\Omega_j}$ for L_{Ξ_j,Ω_j}
21: update $U^{(j)}_{\Xi_j,\Xi_j} \longleftarrow F^{(j)}_{\Xi_j,\Xi_j} - L_{\Xi_j,\Omega_j} L^T_{\Xi_j,\Omega_j}$
22: mark L_{Γ_j,Ω_j} as *evictable*
23: return $U^{(j)}_{\Xi_j,\Xi_j}$

elimination tree. The algorithm then constructs a symmetric $(\omega_j + \xi_j)$-by-$(\omega_j + \xi_j)$ dense matrix $F^{(j)}$ called the *frontal matrix* of j. We initialize $F^{(j)}$ to zero and add to it the submatrix $A_{\Omega_j \cup \Xi_j,\Omega_j}$. The update blocks of the children are then added one by one to $F^{(j)}$ by aligning the columns of their update block to the columns of $F^{(j)}$ in an operation called *extend-add*. Next, the first ω_j columns of $F^{(j)}$ are factored to form $L_{\Omega_j \cup \Xi_j,\Omega_j}$. Finally, the contribution block is computed using the formula $U^{(j)}_{\Xi_j,\Xi_j} = F^{(j)}_{\Xi_j,\Xi_j} - L_{\Xi_j,\Omega_j} L^T_{\Xi_j,\Omega_j}$. The recursion starts at the root of the elimination tree.

Algorithm 1 is a pseudo-code description of **ooc-mf-factor**, a generic OOC extension to the multifrontal method. This is not a complete algorithm, but rather a template for different realizations of OOC multifrontal algorithms.

Out-of-core multifrontal algorithms exploit the fact that there is a potentially large gap between the moment certain data items are computed and the moment they are used. For example, after the factor block has been computed and used to create the corresponding contribution block, it is not used again during the factorization. A contribution block is not used until the factorization of its siblings has been finished. These data items can be *evicted* from main memory. When an evicted item is needed again the algorithm has to *retrieve* it. The fact that a data item (factor block or contribution block) is not going to be used soon does not mean the data item must be evicted. Instead, **ooc-mf-factor** simply marks which data items are *evictable*. A realization of the algorithm specifies how evictable data items are managed. The term "manage" here encapsulates a very wide range of possible strategies. For example, as we will see in Section V, our algorithm moves evictable contribution blocks in an hierarchy of caches before they are actually evicted (if the algorithm runs out of space in main memory). Another possible strategy is to compress the data or even reduce accuracy when writing data to the secondary storage. The analysis in Section IV-E suggest that this can result in a big reduction in I/O volume and compute time.

It is sometimes useful to reason about the most aggressive realization of **ooc-mf-factor**, which we call **aggressive-ooc-mf-factor**. In **aggressive-ooc-mf-factor**, data blocks are always evicted immediately when they are marked evictable. That is, lines 9 and 22 become write (to secondary storage) and free (main memory) operations, and line 15 becomes allocate (memory) and read (from secondary storage) operation. **aggressive-ooc-mf-factor** has the lowest possible memory footprint, so it is the most robust realization of **ooc-mf-factor**, but it also has the highest possible I/O volume (for a given set of supernodes).

Another important realization of **ooc-mf-factor** is to write only factor data to secondary storage, and to do so immediately each time a factor block has been formed. That is, lines 9 and 15 become a no-operation, and line 22 becomes a write+free operation. We call this realization **discardL-ooc-mf-factor**. We call a realization of **ooc-mf-factor** that follows the pattern of **discardL-ooc-mf-factor** in writing all factor data immediately to secondary storage, but allows eviction and retrieval of contribution blocks, an *extension of* **discardL-ooc-mf-factor**.

It is worth reminding the reader that the Cholesky factorization is only a part of the solution process. Once we have the factorization $A = LL^T$, two additional triangular solves need to be performed: forward solve ($Ly = b$), and backward solve ($L^T x = y$). In an OOC solver, during the forward and backward solve, we need to retrieve factor blocks that were evicted from memory.

IV. Analysis of Data Movement Costs

In this section, we analyze the data-movement costs of an OOC multifrontal algorithm. We use the insights gained to design our algorithm, which we present in Section V. To simplify the analysis, we disregard the storage requirements for non-numeric data.

A. Effect of I/O on Shared-Memory Out-of-Core Factorization

The most important concern of an OOC solver is the cost of I/O operations, which are, by far, the most expensive operations in an OOC solver. Typically, writes and reads from secondary storage are at least a few orders of magnitude slower than main memory write and reads, which, in turn, are a few orders of magnitude slower than CPU operations. The gaps continue to increase every year. Furthermore, access to secondary storage typically involves physical hardware which

introduces serialization that might affect the speedup realizable from parallelism.

We now make a couple of observations regarding writing to and reading from secondary storage in a shared-memory parallel OOC linear solver. First, in most practical situations, access to secondary storage is essentially sequential. For example, the head of a magnetic disk can physically be at only one place at a time. Parallel file systems, like Parallel Virtual File System (PVFS) and General Parallel File System (GPFS), do not solve this problem for shared-memory machines. These are designed to support clusters by allowing data to be accessed over multiple machines at once. Increased I/O performance is achieved by splitting files over multiple disks, and reading and writing these blocks in parallel. While these features can be very useful for an OOC solver, they do not change the fact that reading two files in parallel is not faster than reading them one after the other. This is also true for other types of secondary storage, such as solid-state virtual disks, in the context of a shared-memory machine.

Accessing the secondary storage usually involves high latency costs. However, the latency cost can be negligible when writing or reading a large block of data. The nature of the data generated by OOC linear solvers enables them to use buffering to group small read and writes in order to avoid large latency costs. Since this technique is well known and standard, we generally ignore latency costs in our analysis. However, it is important to stress that implementing some buffering is essential, and indeed our code implements it where appropriate.

Based on these observations, we choose the following simplified communication model to analyze data-movement costs in an OOC solver. Memory is divided into two components: *main memory* and *secondary storage*. Both are shared. No data is implicitly transferred between the two levels; only explicit writes and reads to secondary storage are permitted. Main memory is accessed at no cost (relative to the cost of accessing the secondary storage), and in a non-blocking manner. Access to the secondary storage is sequential and blocking, and the I/O cost is proportional to the volume (number of words) of the data written or read (i.e., the latency cost is disregarded).

For added clarity, we now make a distinction between two terms that we use extensively in the remainder of the paper: *I/O volume* and *communication* or *data-movement costs*. *I/O volume* is the total number of words written to secondary storage or read from it. By *data-movement costs*, we refer to the part of the running time that can be attributed to I/O. In our model there is *no* constant that relates the two, since data-movement cost also includes time spent on waiting to get access the secondary storage. Appropriately, we call our model a *communication model* (and not I/O-model) since it describes how running time is affected by I/O.

Other communication models have been studied in the literature. Ballard et al. [8] suggest a sequential model in which a message of k words can be transferred between fast and slow memory in time $\alpha + \beta k$, where α is the latency and β is the inverse bandwidth. This model is designed for analyzing data transferred between cache levels in a sequential algorithm, and is not well suited for analyzing a parallel algorithm with sufficient buffering. Another widely used model is the *two level I/O model* or *disk access machine* [1], [9]. In this model, records (words) are transferred in blocks of size B, where P blocks can be transferred concurrently. This model is designed for I/O-complexity analysis, so it concerns solely with counting the number of blocks transferred between the secondary storage and main memory, and does not specify the interaction between computation and I/O.

B. I/O Complexity of the Out-Of-Core Multifrontal Method

The multifrontal method prompts the use of large blocks when calling the BLAS, which is particularly desirable in a parallel setting. However, Rotkin and Toledo [29] observed that OOC multifrontal solvers can potentially have a poor computation-to-I/O ratio. Consider **aggressive-ooc-mf-factor**. Processing supernode j requires reading/writing $\Theta((\omega_j+\xi_j)^2)$ floating point numbers from/to the secondary storage, but only $\Theta(\omega_j(\omega_j + \xi_j)^2)$ floating-point operations. If $\omega_j \ll \xi_j$ we have a poor computation-to-I/O ratio. This poor ratio creates a bottleneck: in our model writing to secondary storage is not only slow, but also sequential. Rothberg and Schreiber [28] observed this bottleneck in practice for matrices arising in linear-programming codes. However, it is unclear whether it occurs for matrices from other applications.

We try to answer this question in two ways. From a theoretical standpoint, consider an $N^{1/3} \times N^{1/3} \times N^{1/3}$ 3-D grid with supernodes that are "+" separators (a separator formed by three orthogonal planes partitioning the grid into eight equally-sized subgrids), where partitioning is carried out until the finest level. This setup is a good model for a multifrontal factorization of matrices arising in many, but not all, applications. A poor computation-to-I/O ratio will occur for supernodes near the bottom of the elimination tree (these supernodes have tall-and-skinny factor blocks), but these supernodes account for only a very small portion of the overall I/O volume and computational cost of **aggressive-ooc-mf-factor**. For a supernode j near the top of tree that corresponds to a separator of a subgrid of size $K^{1/3} \times K^{1/3} \times K^{1/3}$, we have $\omega_j = \Theta(K^{2/3})$ and $\xi_j = \Theta(K^{2/3})$. Supernodes near the top of tree dominate the I/O volume and computational cost, a fact that can be used to show that for the model problem, **aggressive-ooc-mf-factor** will perform $\Theta(N^2)$ floating-point operations, but will transfer only $\Theta(N^{4/3})$ words between main memory and secondary storage. Thus, we see that for the model problem, the multifrontal method does not suffer from a poor computation-to-I/O ratio.

Grigori et al. [15] proved a lower bound on the communication volume between fast and slow memory of the Cholesky factorization of the same model problem. They assumed that contribution blocks are copied to slow memory once they are formed, and copied back in when they are needed. Once factor blocks have been computed they, too, are copied to slow memory. So, if fast memory is main memory, and slow memory is secondary storage, their setup corresponds

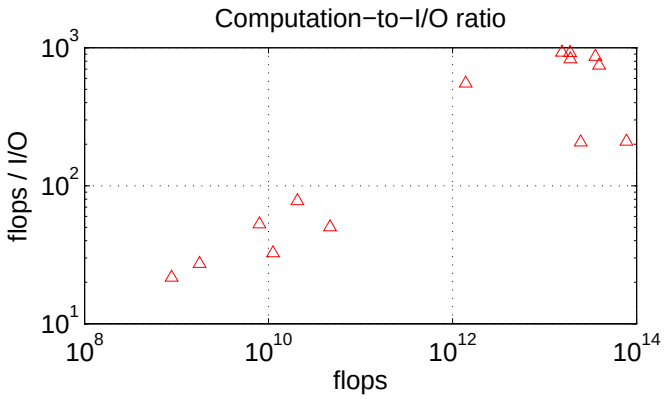

Figure IV.1. Ratio between floating point operations and floating points written to/read from secondary storage by **aggressive-ooc-mf-factor** on a variety of matrices.

to **aggressive-ooc-mf-factor**. They show that if the amount fast memory is not too large, i.e. $M = O(N^{2/3})$ where M is the number of words in fast memory, then a lower bound on the communication between slow and fast memory is $\Omega(N^2/\sqrt{M})$. The minimum amount of main memory required to run **aggressive-ooc-mf-factor** is $\Omega(N^{2/3})$. So by setting $M = \Theta(N^{2/3})$, we find that **aggressive-ooc-mf-factor** is asymptotically optimal in the I/O volume it produces when memory is set to the minimum required.

We supplement the preceding analysis by examining the actual computation-to-I/O ratio that occurs on all matrices in our test-suite (matrices in Table I) and a variety of matrices from the University of Florida Collection[1] [11]. For each matrix, we computed the ratio of the number of floating point operations required to factor the matrix with the column ordering and supernode division used by our solver (WSMP), to the number of floating points written to and read from secondary storage by **aggressive-ooc-mf-factor**. Figure IV.1 shows computation-to-I/O ratio versus the amount of floating operations in log-log scale. We see that as it becomes more and more expensive to factor the matrix, the I/O volume becomes smaller relative to the number of Flops. Hopefully, the data-movement costs become small too.

We therefore conclude that there seems to be a reasonably large class of interesting matrices with a good computation-to-I/O ratio for the multifrontal method, something that our experiments presented in Section VI further confirm.

C. Lowering Data-Movement Costs

Although the I/O volume is asymptotically smaller than the computational cost in our model problem even for the most I/O-expensive realization of **ooc-mf-factor** (**aggressive-ooc-mf-factor**), in practice, I/O is so much more expensive than computation that data-movement costs are still significant even for the largest matrix in our test suite. Furthermore, the access to secondary storage is sequential in our model; so even a relatively small I/O volume (say, accounting for 10% of the serial running time), might cause significant data-movement costs when more computational cores are used.

One straightforward strategy to reduce the data-movement costs is to minimize I/O volume. Agullo [2] analyzed the I/O volume of extensions of **discardL-ooc-mf-factor** (i.e., factor blocks are written to secondary storage as soon as they are computed) which allow partial eviction of contribution blocks (that is, the data of a contribution block can be split between main memory and secondary storage). He shows that once the ordering of the supernodes and allocation scheme[2] has been fixed, minimum I/O volume is attained if the least-recently used contribution block data is written to secondary storage whenever the solver runs out of memory. Agullo et al. [5] studied the effect of supernode ordering and allocation scheme on the I/O volume, under the same assumptions, and suggested an allocation scheme to minimize it. Both analyses focus solely on the sequential case.

Clearly, extensions of **discardL-oof-mf-factor** can be wasteful, and do more I/O than necessary. A simple example is when the total size of active contribution blocks (i.e., contribution blocks that were already computed but not yet assembled into their parent) never exceeds the size of available main memory (that is, main memory that we have in excess of what needed to keep the largest frontal matrix). Even if at some point, the total size of active contribution blocks exceeds available memory, the total size of active contribution blocks will probably drop below available memory at a later point in the factorization. Clearly, it is wasteful not to fill the excess space with factor block data that was created after that point.

Our analysis and our communication model suggest that focusing solely on reducing I/O volume is not the best strategy for a parallel solver. Recall that in our communication model, all access to secondary storage is serial. This is less likely to reduce speedups if the computation-to-I/O ratio is large, since most of the time is spent on computation so a write or read will most likely occur when other cores are busy performing some computation. The analysis in Section IV-B suggests that for the factorization phase, we have a large computation-to-I/O ratio. However, in the forward and backward solve phase, the computation-to-I/O ratio is no longer small. In fact, for extensions of **discardL-oof-mf-factor**, the computation-to-I/O ratio in the backward and forward phase is $\Theta(1)$. Therefore, threads in the parallel solution phase are more likely to be slowed down by I/O than in the factorization phase. The last observation holds to a lesser degree in a sequential run as well, or inside a single thread in a parallel run. Using buffering, I/O can be interlaced with computation, and as long as there is enough computation, the data-movement costs will be small.

To summarize, increasing I/O volume can actually lower the data-movement overhead in a parallel multifrontal solver by shifting part of the I/O burden from the solution phase to the

[1]Matrices included from the University of Florida Collection: cfd1, cfd2, ldoor, msdoor, nd24k, oilpan and vanbody.

[2]In order to streamline this paper, we avoid discussing allocation schemes, since they do not affect the data-movement costs introduced by parallel processing (beyond the fact that they reduce I/O), which is the main focus of our analysis. Agullo et al. [2], [5] discuss allocation schemes in detail.

more forgiving factorization phase. Therefore, up to a certain degree, it may be useful to keep the factor blocks in the main memory at the cost of increasing the I/O volume associated with the contribution blocks.

D. Trade-off Between Parallelism, Memory-Usage, and I/O

A straightforward realization of **ooc-mf-factor** processes frontal matrices in main memory, and writes/reads factor blocks and contribution blocks as one unit. Such a realization has, for each matrix, a minimum memory requirement, which is the maximum space required to keep any possible combination of blocks that have to be in main memory at the same time. An upper bound on this quantity is computable from the elimination tree. A more robust algorithm that splits frontal matrices between main and secondary memory, is possible [28]. However, the additional robustness will be useful only for matrices with memory requirements so high that even **aggressive-ooc-mf-factor** will fail on them, and very large I/O volume will be necessary. It is unlikely that we can gain a reasonable speedup for such matrices, so we do not consider these realizations further since our aim is to analyze costs in a practical parallel multifrontal solver.

It is easy to see that the minimum memory requirements will typically grow when more than one computational core is used. Some type of parallel processing may not affect the minimum memory requirements (e.g., parallel processing used to accelerate the BLAS operations only). However, if two supernodes are factored concurrently, then we need to keep both frontal matrices in main memory simultaneously. Thus, by attempting to exploit more parallelism, we are in effect increasing the minimum memory requirements. The increase in minimum memory usage leaves us with less space to store factor blocks and/or contribution blocks in main memory. Less space translates to more data sent to secondary storage and back, thereby increasing I/O volume and data-movement costs. In our model, I/O is serial, so we are, in effect, reducing our ability to exploit parallelism. Therefore, if the solver is too greedy in trying to exploit parallelism, it may end up losing more than it gains. A multifrontal OOC solver needs to strike a delicate balance between the added benefit of using parallelism, and the additional data-movement costs associated with exploiting parallelism.

E. Accuracy and I/O Trade-off

Storing the factor blocks in single instead of double precision can significantly reduce I/O volume due to writing and reading the factor blocks, and with it the associated data-movement costs. There is a tradeoff between accuracy and I/O volume. Reducing accuracy does not have to be as aggressive as reducing double precision numbers to single precision; it can be reduced to any number of user-defined bits. Under the right circumstances, it is possible to accelerate OOC triangular solves to an extent that iterative refinement may be used to recover all or most of the lost accuracy, while still decreasing the overall running time. A similar approach has been shown to be effective in dense solvers in a different context [31].

If c is the data-compression ratio ((number of bits used to store a floating-point number)/64) and r is the fraction of L that can be stored in main memory without compression, then using compression would reduce the I/O volume by a factor of $(c-r)/(1-r)$. For $c \leq r$, I/O during the triangular solves can be completely eliminated; clearly, this is practical only for sufficiently large values of r. A more feasible condition of $c \leq (r+1)/2$ would permit the I/O volume to drop by more than half, thus permitting at least one step of iterative refinement, if required, at no additional cost.

Based on this analysis, we are currently investigating suitable lossy compression schemes and will report the results in a future publication.

V. Our Algorithm

Based on the analysis in Section IV, we devised techniques to reduce the data-movement costs and manage the tradeoffs. We implemented these techniques in a new OOC linear solver for SPD matrices. In this section, we describe the algorithmic details pertaining to the OOC functionality.

A. General

Our OOC solver has been implemented on top of the existing parallel in-core solver for SPD systems in WSMP [16]. Our algorithm processes frontal matrices in main memory, and and our current implementation writes/reads individual factor and contribution blocks as indivisible units. As explained in Section IV-D, a more robust algorithm is possible [28], but is unlikely to deliver decent speedups for matrices that require the extra robustness. Since our code was designed to be used primarily as a parallel code, it does not make sense to optimize it for such matrices. Our algorithm stores only numeric data, that is the factor blocks and contribution blocks, in secondary storage. Non numeric data are always stored in main memory. Typically, most of the non-numeric data is symbolic data whose storage requirements are much smaller.

WSMP exploits parallelism both at the fine-grain level of the dense BLAS operations, and at the coarse-grain level of the elimination tree. The coarse-grain parallelism is exploited by splitting the elimination tree into disjoint subtrees, each one factored by a single task in a dynamic scheduling framework. The elimination tree is broken into more tasks than the number of physical cores in order to achieve better load-balancing through work-stealing. Henceforth, the word *task* refers to these coarse-grain tasks.

B. Memory Management

Our algorithm exploits the main memory using a smart memory management scheme. Following the observations in Section IV-C, the goal of the memory management scheme is both to reduce the amount of data transferred between main memory and secondary storage throughout the linear solve (that is, factorization and triangular solves), and also to keep as much factor data in main memory as possible. In a parallel factorization, it is impossible to predict exactly how memory requirements will develop during the factorization. Therefore,

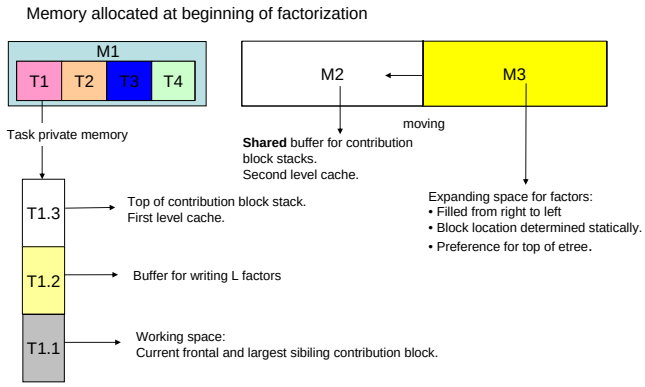

Figure V.1. Memory management used by our solver.

we opt for a dynamic and heuristic memory management scheme.

Figure V.1 shows a schematic diagram that depicts our memory management scheme. Our solver allocates all the main memory that it will use at the start of the factorization. The allocated memory is split into two blocks; in Figure V.1 these blocks are displayed as the top rectangles. The first block, $M1$, contains task private areas. The second block holds two memory areas: $M2$ and $M3$. $M2$ is a second level buffer for contribution blocks, and $M3$ is storage space for factor blocks. The size of $M1$ is fixed and is determined during the symbolic phase. While the total size of $M2+M3$ is also fixed (and determined during the symbolic phase), the relative sizes of $M2$ and $M3$ are not fixed. $M3$ starts empty and grows at the expense of $M2$ during the factorization. The total size of $M1+M2+M3$ is derived from the user-supplied information on the total amount of main memory available.

Each task i has its own private area Ti in $M1$. For more efficient utilization of main memory, our implementation permits Ti to subsume $T(i{+}1)$, $T(i{+}2), \ldots$ etc. after adjacent tasks terminate as the computation moves closer to the root of the elimination tree. In other words, the boundaries of Ti are dynamic and depend on whether or not adjacent areas are being used. Each task private area is further split into three parts. The first part, $Ti.1$, is a workspace for processing the current frontal matrix. This workspace stores the current frontal matrix. It is also used to load contribution blocks from secondary storage before they are assembled into the front. The second part, $Ti.2$, is a buffer for writing factor blocks to secondary storage. As explained in Section IV-A, grouping small I/O operations by buffering is crucial for avoiding large latency costs. Our experiments suggest that a 8MB per task buffer is sufficient. It is worth noting that the operating system performs a similar buffering. Nevertheless, we observed that using additional software level buffering improved performance. The third part, $Ti.3$, is a cache for contribution blocks. Once a contribution block has been computed, our algorithm places it in a stack that is logically held on secondary storage. In practice, it is much more efficient to keep contribution blocks in memory if there is space for them. We use a two-level cache for storing contribution blocks in memory. $Ti.3$ is the first level task-private cache, where we store the top of the task's contribution block stack. The top of the stack is the one accessed most frequently. The minimum size of each $Ti.3$ is 16MB, but can be larger if the adjacent space of inactive tasks can be included.

Memory $M2$ is used as a second level cache for contribution blocks. When the size of a contribution block stack grows beyond the size of the first level cache $Ti.3$, the entire $Ti.3$ block is transferred to $M2$. When $M2$ has filled up as well, we transfer data to secondary storage to free space for newer data. All tasks share $M2$ and compete for space in it, so access to it is protected by a lock. A task can evict contribution blocks from other tasks, but eviction is almost always of least-recently added contribution blocks (bottom of the stacks). The only case in which we evict a contribution block that is not the oldest one in $M2$ is when we encounter a contribution block that is larger than the entire space in $M2$. In this case, even if we evict all contribution blocks currently in $M2$, the new contribution block will have to be evicted. Therefore, we evict this block to secondary storage without evicting all the blocks in $M2$. However, since we want to keep only a single continuous part of each stack in $M2$, we must evict all contribution blocks that originated from the task that created the new contribution block, even if there are older contribution blocks in $M2$.

$M3$ stores factor blocks. Our algorithm determines during the symbolic analysis which factor blocks are kept in $M3$, and which ones are evicted. The next subsection describes how symbolic analysis determines that. The symbolic analysis also predetermines the exact location of each block that is kept in $M3$. Because the location of the factor blocks is determined before the start of the factorization, there is no contention between threads in copying data to $M3$. When a factor block is computed, it is either written to secondary storage or stored in $M3$.

The size of $M2$ shrinks during the factorization: when a factor block is stored in $M3$, the size of $M2$ may need to shrink. In some cases this shrinkage causes the size of $M2$ to drop below the amount of data inside it. In that case, we evict least-recently added contribution blocks until the size of kept contribution blocks has shrunk enough.

C. Mapping Factor Blocks to Memory

The analysis in section IV-C suggests that at least some of the factor data should be kept in main memory, but it is also important to avoid increasing the data-movement costs associated with the contribution blocks by too much. We deduce the following general guidelines for assigning factor blocks to $M3$:

1) We want to assign as much factor block data to $M3$ as possible.
2) It is desirable for $M2$ to shrink as slowly as possible.

Condition (1) is, by itself, NP-hard (by reduction to KNAPSACK). Condition (2) cannot be met exactly in a parallel solver, since it is impossible to tell the exact order in which

Algorithm 2 Assigning factor blocks to memory.

1: **Input:**
2: Elimination tree T with root r ; $\mathrm{dist}_T(u, v)$ is distance in T between u and v.
3: Address space $[L, U)$ for assigning factor blocks.
4: Supernodes sizes: $\omega_0, \ldots, \omega_{S-1}$ and $\xi_0, \ldots, \xi_{S-1}$.
5: **Output:**
6: Locations for factor blocks $l_1 \ldots, l_S \in [L, U) \cup \{\infty\}$.
7:
8: Find an ordering of supernodes $j_0, \ldots, j_{S-1}$ such that $\mathrm{dist}_T(j_i, r) \leq \mathrm{dist}_T(j_{i+1}, r)$.
9: $e \longleftarrow L$
10: **for** $i = 0$ **to** $S - 1$ **do**
11: $\quad s_i \longleftarrow \frac{\omega_{j_i}(\omega_{j_i}+1)}{2} + \omega_{j_i}\xi_{j_i}$
12: $\quad$ **if** $s_i \leq U - e$ **then**
13: $\quad\quad l_{j_i} \longleftarrow e$; $e \longleftarrow e + s_i$
14: $\quad$ **else**
15: $\quad\quad l_{j_i} \longleftarrow \infty$
16: $\quad$ **end if**
17: **end for**
18: **for** $i = 0$ **to** $S - 1$ **do**
19: $\quad l_{j_i} \longleftarrow l_{j_i} + U - e$
20: **end for**

factor blocks will be ready. Therefore, we opt for heuristics for mapping factor blocks to main memory.

Algorithm 2 gives a pseudo-code description of our strategy. We use a greedy approach to address criterion (1). We address criterion (2) by prioritizing for allocation blocks generated near the end of the factorization, since that will cause most of $M2$'s shrinkage to occur near the end of the factorization. The exact order in which blocks are computed in parallel cannot always be deduced, so instead our algorithm begins by building an approximation to the order in which factor blocks are computed (line 8). It does so by assuming that all factor blocks at distance d from the root are computed before factor blocks at distance $d - 1$. Within a level, we order the factor blocks arbitrarilyy. We now consider the factor blocks in a reverse order to their assumed computation time, that is, the top of the elimination tree first. For each factor block, we greedily check if there is still enough space to store it (line 12). If there is, we map it to memory so that it will occupy the end of the address space that has not been assigned yet (lines 13 does the initial assignment, and lines 18-20 move it to higher addresses). If there is not enough space, then the factor block is mapped to secondary storage (line 15).

The forward and backward solve phase require considerably less memory than the factorization phase. This implies that there is space to store additional factor blocks after they have been loaded to memory. After the factorization is complete, our algorithm uses a greedy postprocessing heuristic to try to assign some of the factor blocks that are on the secondary storage to the now free space in $M1$ and $M2$. During the forward solve, a factor block that is on the secondary storage is loaded into the main memory when it is needed for computation. After this block is used, it is discarded if it was not assigned to main memory by the postprocessing heuristic; otherwise, it is kept in its newly designated location in the main memory. Subsequent triangular solves will omit reloading this block.

D. Adaptive Task Allocation

As the number of computational cores and consequently, the number of parallel tasks increases, the minimum amount of main memory required for OOC factorization also grows. As explained in Section IV-D, the increase in the minimum memory requirement can inhibit parallelism, which is the opposite of what we desire from increasing the number of cores. In the terminology of Section V-B, increasing the number of tasks increases the size of $M1$ at the expense of $M2$ and $M3$, which results in more I/O.

We modified WSMP's regular task allocation algorithm to address this issue. First, we summarize how WSMP normally (i.e., in-core [16]) splits the elimination tree into tasks—a process we call task allocation. The allocation process starts at the root of the elimination tree with a number of tasks that is a constant times the number of cores to be employed. The allocation proceeds recursively, splitting the number of tasks among subtrees at each level in proportion to the work associated with each subtree. The process continues until each subtree is assigned a single task.

We implemented the following measures to avoid slow down due to high minimum memory requirements. First, we reduced the constant that determines the number of tasks at the root, so that the OOC solver is less aggressive in trying to exploit parallelism. The second technique is *pushing down of parallelism*, and is illustrated in Figure V.2. Instead of splitting the tasks, we assign all the tasks to all the subtrees at a predetermined number of levels close to the root of the elimination tree. The impact of this allocation is that these subtrees are not processed concurrently, but one after the other instead. Each subtree, however employs all the tasks assigned to it. Pushing down of parallelism reduces the granularity of parallelism. The number of levels by which parallelism is pushed down, if at all, is heuristically determined based on the user-indicated amount of main memory available, the sizes of L and the largest contribution block, and the number of cores to be employed. Figure V.2 illustrates pushing down parallelism by 2 levels.

Pushing parallelism down in the elimination tree reduces the amount of main memory required to concurrently store large contribution blocks associated with the supernodes near the top of the elimination tree. The relatively small impact on parallel efficiency is more than made up by the reduction in I/O.

VI. Experiments

We now describe the experimental results obtained with the new OOC solver in WSMP, as well as comparisons to two other OOC solvers: the sequential solver HSL MA77 [26],

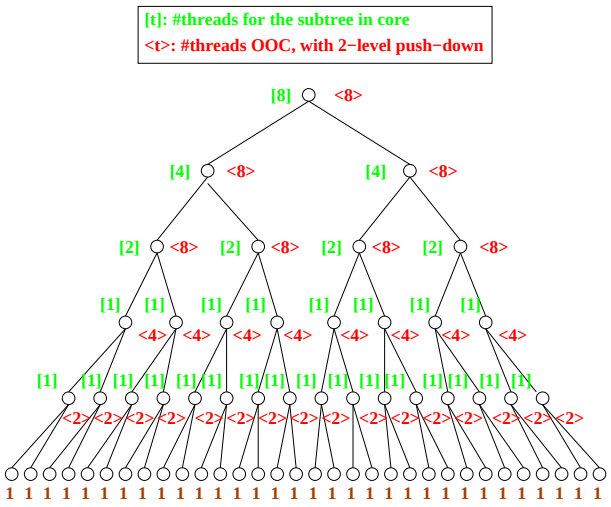

Figure V.2. An illustration of adaptively pushing parallelism down the elimination tree in parallel sparse OOC Cholesky factorization.

and the parallel solver Intel MKL PARDISO [18]. We chose MA77 and PARDISO because they are the most recent OOC codes in their respective categories; i.e., multifrontal and left-looking. Comparison to PARDISO is also important since it is the only other fully OOC shared-memory parallel solver.

All the experiments reported in this paper were performed on a 32GB machine with two 2.33GHz quad-core Intel E5410 running Linux 2.6. Secondary storage is a 10K RPM SAS drive. We linked WSMP and MA77 with the sequential version of MKL 10.3 (PARDISO is part of MKL so the BLAS is built-in). Before running, we set the system swappiness to 0, by writing 0 to **/proc/sys/vm/swappiness**. With a lower swappiness value, the kernel is less apt to swap out allocated memory pages. For an OOC solver, swapping out of allocated memory is undesirable since the eviction of data is controlled by the solver directly.

We generally restricted the in-core memory to 20 GB, unless the symbolic data was too large and memory had to be restricted even more. That is, in WSMP the size of the combined size of $M1$, $M2$ and $M3$ is 20 GB. In MA77, the buffer for the virtual memory is 20 GB. In PARDISO, we set MKL_PARDISO_OOC_MAX_CORE_SIZE to 20 GB.

PARDISO does not have any additional parameters that need tuning. MA77 has a few parameters that need tuning. First, there is the size of pages in the virtual memory buffer, which is controlled by `buffer_lpage`. Second, the memory allocated virtual memory buffer is split between two different buffers: one for real numbers and one for integer numbers. Once the page size and the size of the overall buffer is determined, the ratio of the split can be controlled using `buffer_npage`. We tuned MA77 by first fixing the page size and varying the split between the two buffers. Once we found the best split, we varied the the page size to find the best setting for it.

Table I contains a list of the matrices in our tests. The matrices were extracted from some of the real applications that use WSMP. The suite is rather small because there are few large matrices publicly available. Furthermore, the runs are very long, so a large data set is not practical. The matrices are divided into two groups, separated by a double horizontal line in Table I. The first group consists of matrices that can be factored in-core in WSMP, but only in the sequential case. The second group consists of matrices which require the OOC functionality for successful factorization by WSMP even on a single core. Matrices were preordered to reduce fill-in using Metis 4.0.3 [19]. WSMP and MA77 succeeded in solving all matrices in the test set. PARDISO failed on **box50** ("ordering problem") and **xxjst** ("out of memory").

Table I
MATRICES IN OUR TEST-SUITE. ALL NUMBERS ARE DIVIDED BY 10^6.

Name	N	#nnz(A)	#nnz(L)	Application/Origin
algor-big	1.1	42.7	2524.2	Static stress analysis
ten-b	1.4	54.1	3134.2	3-D metal forming
cube140	2.8	10.9	3387.1	$140 \times 140 \times 140$ Grid
box50	1.6	61.9	4092.1	Biomechanical engineering
sgi-3M	3.1	130.2	6198.7	Structural analysis
xxjst	7.4	291.4	8033.9	Structural engineering
cube170	4.9	19.6	7517.1	$170 \times 170 \times 170$ Grid

Figure VI.1 compares WSMP's total running time (both factor and solve phases) with the corresponding running times of MA77 and PARDISO for the matrices in Table I. For the two graphs on a single core (left and middle graph), we plot the in-core time ratio where applicable, along with the OOC time ratio. On eight cores (right graph), PARDISO was able to factor in-core several matrices for which WSMP could only factor OOC. In these cases, we also plotted WSMP's OOC time vs. PARDISO's in-core time. With the exception of a single case (**algor-big** on 8 cores) OOC WSMP is faster than OOC MA77 and OOC PARDISO. For both MA77 and PARDISO, if we compare the OOC vs. OOC ratio to the IC vs. IC ratio, we see that WSMP compares more favorably in the OOC setting than in the in-core setting (the only exception being **algor-big** for WSMP compared to PARDISO). This suggests that WSMP's advantage in the OOC setting is the result of the better data-management scheme presented in this paper, and not only due to a better in-core baseline. Since, PARDISO uses a left-looking algorithm, its memory requirements are smaller. As a result, in some cases, it is able to factorize in-core matrices for which WSMP requires an OOC solver. However, we see (rightmost graph in Figure VI.1) that for these cases, OOC WSMP is faster than in-core PARDISO. This is due to the nature of scaling of OOC PARDISO, as explained below.

Figure VI.2 shows the results of a scalability study using matrices derived from 3-D grids so that we can control problem size without altering other aspects of the problem. In the left graph, we examine if various codes are able to keep the same asymptotic behavior when the grid becomes larger. OOC solvers are able to solve larger systems than in-core solvers, but this is worthless if their performance degrades too much. It is well known that solving an $n \times n \times n$ 3-D grid system requires $\Theta(n^6)$ operations. Therefore, we expect the running time of a scalable code to be approximately Cn^6 for some suitable constant C. We approximate this constant by

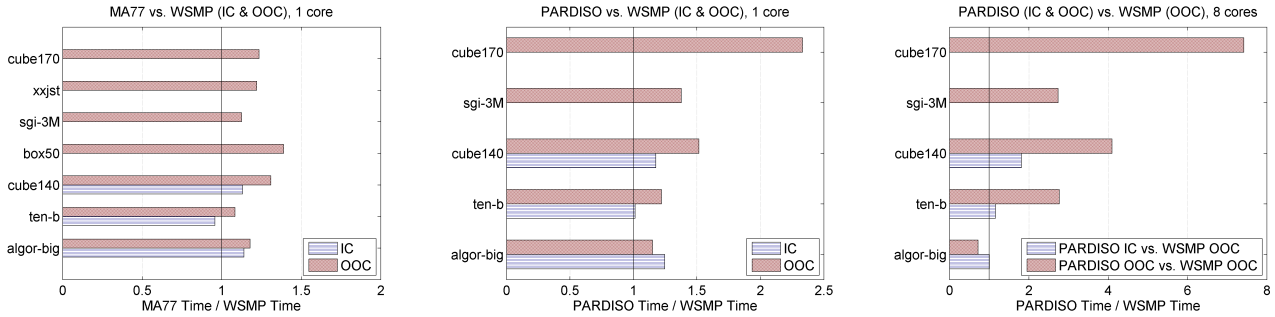

Figure VI.1. Comparison between WSMP, MA77, and PARDISO factorization and solve times. The longer the bar, the faster WSMP is versus the other code.

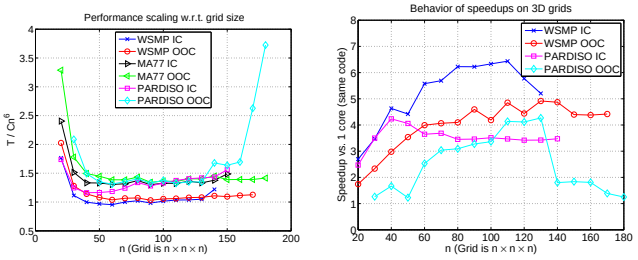

Figure VI.2. Scalability study using 3-D grids. The left graph plots the ratio of serial running time (T_n) for $n \times n \times n$ grids to Cn^6, where $C = T_{80}^{WSMPIC}/(80)^6$, against n. The right graph examines the speedup with 8 cores for various grid sizes.

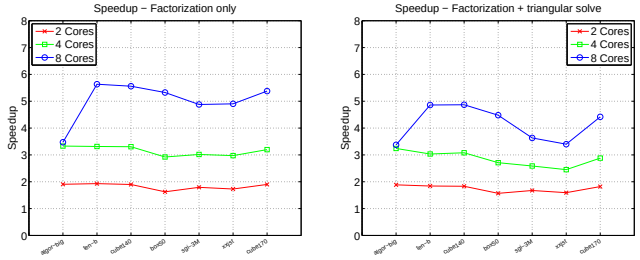

Figure VI.3. Speedups of OOC WSMP with 2, 4, and 8 cores.

measuring WSMP's running time for $n = 80$. A scalable OOC solver need not only have a running time close to its in-core counterpart, but should also be able factor larger grids with the same asymptotic behavior (same ratio between running time and Cn^6). In the left graph, we plot the running time divided by Cn^6 for various codes. We see that in-core WSMP is able to keep the ratio near one until it runs out of memory. Out-of-core WSMP is able factor larger grids while keeping the ratio still constant, but slightly greater than the in-core ratio. The same is true with larger constants for in-core and OOC MA77, and for in-core PARDISO. While OOC PARDISO is able to factor some matrices that OOC WSMP fails on, the ratio deteriorates rapidly, starting with grids that are small enough for WSMP to handle with a constant ratio. We see that WSMP and MA77 are able to scale for larger grids, but WSMP is faster. PARDISO is able to solve even larger grids, but it does not scale.

The right graph in Figure VI.2 examines how the speedup of WSMP and PARDISO behaves on eight cores as the grid grows. Again, a scalable code should be able to keep the same speedup behavior for larger grids than its in-core counterpart. We see that OOC WSMP is able to keep a rather constant speedup longer than in-core WSMP, so its parallel performance in OOC scales well. In contrast, OOC PARDISO's parallel speedup deteriorates as the grid size increases and consequently, more data is exchanged between main memory and secondary storage. This observation seems to corroborate our earlier argument that an increase in I/O in a shared-memory OOC solver can be detrimental to parallelism.

In Figure VI.3, we show WSMP's speedup for two, four and eight cores. For overall solution time, we see a speedup of almost 2 for two cores, around 3 for four cores and between 3.5 and 5 for eight cores. Better speedups are obtained for the factorization phase; approaching almost 6 for eight cores on some matrices. This is not surprising; the triangular solves are I/O-bound and therefore yield little or no speedup.

VII. Concluding Remarks and Future Work

We have studied data-movement costs in a shared-memory parallel OOC sparse solver and have proposed, implemented, and experimentally evaluated techniques for minimizing them. We observed that contrary to conventional wisdom, between the triangular factors and the contribution blocks, it is the triangular factors that should have an higher priority of being kept in main memory. We also observed that unless managed carefully, parallelism in a shared-memory OOC solver can be self-limiting. Accordingly, we devised a memory management scheme for dynamically apportioning the available main memory between contribution and factor blocks, and a task parallelization strategy that self-adapts based on the sizes of required and available main memory. We observed that an OOC sparse solver can benefit from lossy compression, which we plan to incorporate in our solver in the near future.

WSMP's distributed-memory parallel symmetric solver [17] recursively partitions the elimination tree among groups of machine nodes and eventually uses the multithreaded code on (presumably shared-memory parallel) individual nodes. The OOC algorithm proposed here applies directly to the roots of the subtrees assigned to individual nodes, and incorporating it is a natural extension of this work.

REFERENCES

[1] A. Aggarwal and S. Vitter, Jeffrey, "The input/output complexity of sorting and related problems," *Commun. ACM*, vol. 31, pp. 1116–1127, September 1988. [Online]. Available: http://doi.acm.org/10.1145/48529.48535

[2] E. Agullo, "On the out-of-core factorization of large sparse matrices," Ph.D. dissertation, École Normale Supérieure de Lyon, Nov. 2008.

[3] E. Agullo, A. Guermouche, and J.-Y. L'Excellent, "A preliminary out-of-core extension of a parallel multifrontal solver," in *EuroPar'06 Parallel Processing*, 2006. [Online]. Available: http://dx.doi.org/10.1007/11823285_111

[4] ——, "Towards a parallel out-of-core multifrontal solver: Preliminary study," LIP, Tech. Rep. RR2007-06, 02 2007, also appeared as INRIA report 6120. [Online]. Available: http://lara.inist.fr/handle/2332/1111

[5] ——, "Reducing the I/O volume in sparse out-of-core multifrontal methods," *SIAM J. Sci. Comput.*, vol. 31, no. 6, pp. 4774–4794, 2010. [Online]. Available: http://epubs.siam.org/sisc/resource/1/sjoce3/v31/i6/p4774_s1

[6] P. Amestoy, I. Duff, and J.-Y. L'Excellent, "MUMPS: A MUltifrontal Massively Parallel sparse direct Solver," *http://mumps.enseeiht.fr/*. [Online]. Available: http://mumps.enseeiht.fr/

[7] C. Ashcraft and R. Grimes, "The influence of relaxed supernode partitions on the multifrontal method," *ACM Trans. Math. Softw.*, vol. 15, pp. 291–309, December 1989. [Online]. Available: http://doi.acm.org/10.1145/76909.76910

[8] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz, "Communication-optimal parallel and sequential Cholesky decomposition," *SIAM Journal on Scientific Computing*, vol. 32, no. 6, pp. 3495–3523, 2010. [Online]. Available: http://link.aip.org/link/?SCE/32/3495/1

[9] M. A. Bender, G. S. Brodal, R. Fagerberg, R. Jacob, and E. Vicari, "Optimal sparse matrix dense vector multiplication in the I/O-model," in *Proceedings of the nineteenth annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '07. New York, NY, USA: ACM, 2007, pp. 61–70. [Online]. Available: http://doi.acm.org/10.1145/1248377.1248391

[10] *BCSLIB-EXT - award winning sparse-matrix package. http://www.boeing.com/phantom/bcslib-ext*, Boeing, 2003.

[11] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, pp. 1:1–1:25, Dec. 2011. [Online]. Available: http://doi.acm.org/10.1145/2049662.2049663

[12] F. Dobrian, "External memory algorithms for factoring sparse matrices," Ph.D. dissertation, Old Dominion University, Nov. 2008.

[13] F. Dobrian and A. Pothen, "Oblio: Design and performance," in *Applied Parallel Computing. State of the Art in Scientific Computing*, ser. Lecture Notes in Computer Science, J. Dongarra, K. Madsen, and J. Wasniewski, Eds. Springer Berlin / Heidelberg, 2006, vol. 3732, pp. 758–767.

[14] Duff, I. S. and Reid, J. K., "The multifrontal solution of indefinite sparse symmetric linear equations," *ACM Trans. Math. Software*, vol. 9, no. 3, pp. 302–325, 1983. [Online]. Available: http://dl.acm.org/citation.cfm?id=356047

[15] L. Grigori, P.-Y. David, J. Demmel, and S. Peyronnet, "Brief announcement: Lower bounds on communication for sparse Cholesky factorization of a model problem," in *SPAA'10*, 2010, pp. 79–81. [Online]. Available: http://dl.acm.org/citation.cfm?id=1810496

[16] A. Gupta, M. Joshi, and V. Kumar, "WSMP: A high-performance shared- and distributed-memory parallel symmetric sparse linear solver," in *PARA'98 Workshop on Applied Parallel Computing in Large Scale Scientific and Industrial Problems*, B. Kagstrom, J. J. Dongarra, E. Elmroth, and J. Wasniewski, Eds. Springer Verlag, 1998, also available as *RC 22038, IBM T. J. Watson Research Center, Yorktown Heights, NY*.

[17] A. Gupta, S. Koric, and T. George, "Sparse matrix factorization on massively parallel computers," in *SC09 (International Conference for High Performance Computing, Networking, Storage and Analysis)*, 2009.

[18] *Intel Math Kernel Library (Intel MKL) 10.3 Reference Guide*, Intel, Santa Clara, CA, May 2010.

[19] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on Scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998. [Online]. Available: http://link.aip.org/link/?SCE/20/359/1

[20] J. W. H. Liu, "On the storage requirement in the out-of-core multifrontal method for sparse factorization," *ACM Trans. Math. Softw.*, vol. 12, pp. 249–264, September 1986. [Online]. Available: http://doi.acm.org/10.1145/7921.11325

[21] ——, "The role of elimination trees in sparse factorization," *SIAM J. Matrix Anal. Appl.*, vol. 11, pp. 134–172, January 1990. [Online]. Available: http://dl.acm.org/citation.cfm?id=80034.80044

[22] ——, "The multifrontal method for sparse matrix solution: Theory and practice," *SIAM Rev.*, vol. 34, no. 1, pp. 82–109, 1992. [Online]. Available: http://epubs.siam.org/sirev/resource/1/siread/v34/i1/p82_s1

[23] E. G. Ng and B. W. Peyton, "Block sparse Cholesky algorithms on advanced uniprocessor computers," *SIAM J. Sci. Comput.*, vol. 14, pp. 1034–1056, September 1993. [Online]. Available: http://epubs.siam.org/sisc/resource/1/sjoce3/v14/i5/p1034_s1

[24] J. K. Reid, "TREESOLVE, a Fortran package for solving large sets of linear finite-element equations," AERE Harwell, Harwell, UK, Tech. Rep. Report CSS 155, 1984.

[25] J. K. Reid and J. A. Scott, "Algorithm 891: A Fortran virtual memory system," *ACM Trans. Math. Softw.*, vol. 36, pp. 5:1–5:12, March 2009. [Online]. Available: http://doi.acm.org/10.1145/1486525.1486530

[26] ——, "An out-of-core sparse Cholesky solver," *ACM Trans. Math. Softw.*, vol. 36, pp. 9:1–9:33, April 2009. [Online]. Available: http://doi.acm.org/10.1145/1499096.1499098

[27] E. Rothberg and A. Gupta, "Efficient sparse matrix factorization on high performance workstations and exploiting the memory hierarchy," *ACM Trans. Math. Softw.*, vol. 17, pp. 313–334, September 1991. [Online]. Available: http://doi.acm.org/10.1145/114697.116809

[28] E. Rothberg and R. Schreiber, "Efficient methods for out-of-core sparse Cholesky factorization," *SIAM J. Sci. Comput.*, vol. 21, pp. 129–144, August 1999. [Online]. Available: http://epubs.siam.org/sisc/resource/1/sjoce3/v21/i1/p129_s1

[29] V. Rotkin and S. Toledo, "The design and implementation of a new out-of-core sparse Cholesky factorization method," *ACM Trans. Math. Softw.*, vol. 30, pp. 19–46, March 2004. [Online]. Available: http://doi.acm.org/10.1145/974781.974783

[30] R. Schreiber, "A new implementation of sparse Gaussian elimination," *ACM Trans. Math. Softw.*, vol. 8, pp. 256–276, September 1982. [Online]. Available: http://doi.acm.org/10.1145/356004.356006

[31] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra, "Dense linear algebra solvers for multicore with gpu accelerators," in *Proc. of IPDPS'10*, 2010.